The SU User's Manual

Jack K. Cohen & John W. Stockwell, Jr.

Version 1.0: July, 1994

Center for Wave Phenomena
Colorado School of Mines

Gas Research Institute
Chicago, Illinois

Legal Matters

The following items pertain to the CWP/SU software available from the Colorado School of Mines anonymous ftp site at hilbert.mines.colorado.edu (138.67.12.63).

Disclaimer: There are no guarantees, explicit or implicit, made by the Center for Wave Phenomena, the Colorado School of Mines, or any member of the aforesaid organizations, past, present, or future, regarding the accuracy, safety, usefulness, or any other quality or aspect of this software.

Copyright: Copyright (c) Colorado School of Mines, 1994. All rights reserved.

License: Permission to use, copy, and modify this software for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies, and the name of the Colorado School of Mines (CSM) not be used in advertising or publicity pertaining to this software without specific, written prior permission. CSM makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Address questions about these legal statements or about the software to either author:

Jack K. Cohen
Center for Wave Phenomena
Colorado School of Mines
Golden, CO 80401

jkc@dix.mines.colorado.edu
(303) 273-3512

John W. Stockwell, Jr.
Center for Wave Phenomena
Colorado School of Mines
Golden, CO 80401

john@dix.mines.colorado.edu
(303) 273-3049

Contents

Acknowledgments v

1 About SU **1**

2 How to Get a Copy of SU **3**
2.1 Obtaining files by anonymous ftp 3
2.2 Requirements for installing the package 4
2.3 A quick test 5

3 Help Facilities **7**
3.1 self-documentation (selfdoc) 7
3.2 suhelp 7
3.3 suname 8
3.4 sudoc 8
3.5 sufind 9
3.6 sukeyword 10
3.7 Other help mechanisms 11

4 Using SU **13**
4.1 SU and UNIX 13
4.2 Exploring SU 14
4.2.1 Looking for DMO programs 14
4.2.2 Getting information about SU programs 15
4.2.3 Viewing header field definitions 16
4.2.4 Viewing program names 16
4.3 Understanding and using SU shell programs 17
4.3.1 A simple SU processing flow example 17
4.3.2 Executing shell programs 19
4.3.3 A typical SU processing flow 19
4.4 Extending SU by shell programming 21
4.5 Some core programs 25
4.5.1 Examining the trace headers 25
4.5.2 Some common processing programs 25
4.5.3 Some common plotting programs 25
4.6 A brief tour of the source directories 26

5 Frequently Asked Questions **27**
5.1 Installation questions 27
5.2 Data format questions 28
5.3 Tape reading and writing 30
5.4 General 31

6 How to Write an SU Program **33**
6.1 A template SU program 33
6.2 Writing a new program: `suvlength` 36

Acknowledgments

We are pleased to acknowledge that SU is partially supported by the Gas Research Institute. This welcome support comes at a time when SU is achieving worldwide distribution and thus is making greater demands on our resources.

The sponsors of the CWP Consortium Project have long been partners in the SU project and we are pleased to explicitly acknowledge that relationship here. In addition, we wish to acknowledge extra support supplied in the past by IBM Corporation and by the Center for Geoscientific Computing at the Colorado School of Mines during the period when SU was ported to the modern workstation from its previous incarnation on graphics terminals.

There are faculty and students, both at our Center and elsewhere, who have contributed to SU, that it is impossible to list them all here. However, certain people have made such signal contributions that they deserve mention whenever the package is discussed.

Shuki Ronen wrote the first draft of what is now called SU in cooperation with Einar Kjartansson while both were graduate students at Stanford University. In turn, some of the fundamental concepts they implemented were formulated by their mentor, Jon Claerbout, Director of the Stanford Exploration Project. Ronen brought this work to our Center during a two-year stay here and, during this time, aided Cohen in turning SU into a supportable and exportable product.

Chris Liner, while a student at the Center, wrote most of the graphics codes used in the pre-workstation (i.e, graphics terminal) age of SU. Liner's broad knowledge of seismology and seismic processing enabled him to make a positive and continuing influence on the SU coding philosophy.

Dave Hale wrote several of the "heavy duty" processing codes as well as most of the core scientific and graphics libraries. His knowledge of good C-language coding practice helped make our package a good example for applied computer scientists.

Ken Larner contributed many user interface ideas based on his extensive knowledge of seismic processing in the "real world."

John Scales showed how to use SU effectively in the classroom in his electronic text, *Theory of Seismic Imaging*, Samizdat Press, 1994. This text is available from the CWP anonymous ftp site.

Chapter 1

About SU

Seismic Unix (SU) is a self-contained software environment for seismic research and data processing used by exploration geophysicists, earthquake seismologists, environmental engineers, software developers and others. It is used by scientific staff in both small companies and major oil and gas companies, and by academics and government researchers.

The SU package is *free software*, distributed quarterly with the full source code, so that users can alter and extend its capabilities. The package permits the exchange of data according to the industry protocol (SEG-Y). It provides a standard environment for the testing of new processing algorithms. It is easy to use because it does not require learning a special language—its application uses only the standard facilities afforded by the UNIX operating system. Once UNIX shell-redirecting and pipes are mastered, there is no further artificial language to learn. The seismic commands and options can be used as readily as other UNIX commands. In particular, the user can write ordinary UNIX shell scripts to combine frequent command combinations into meta-commands (i.e., processing flows). These scripts can be thought of as "job files."

The seismic processing programs in the package assume that the data are written in SEG-Y format with each trace preceded by an appropriate header. This allows the data information to be read by each program in the processing stream in a consistent manner. The package includes facilities for converting data in several other formats to the SEG-Y format.

The SU user community accesses the software over the Internet using the commonly available "anonymous ftp" facility. In this way, users obtain the software with no constraints on its use. During installation, the user is given the option of sending an electronic mail request to add them to our user group. Members of the user group receive announcements when updates of the package become available. The SU community is now worldwide, spanning six continents, more than 16 countries and over 200 known installations on a variety of hardware platforms ranging from mainframes to workstations and PC's.

Parts of SU originated from software developed by students working with the Stanford Exploration Project at Stanford University. The present package was developed and is maintained at the Center for Wave Phenomena (CWP) at the Colorado School of Mines. CWP is an interdisciplinary (geophysics, mathematics) research and educational program in seismic exploration, supported by 23 companies in the oil and gas industry.

Chapter 2

How to Get a Copy of SU

The SU package contains seismic processing programs along with libraries of scientific routines, graphics routines and routines supporting the SU coding conventions. The package is available by anonymous ftp at the site hilbert.mines.colorado.edu (138.67.12.63). The directory path is pub/cwpcodes. Take the files:

1. README_BEFORE_UNTARRING
2. untar_me_first.xx.tar.Z
3. cwp.su.all.xx.tar.Z

Here the `xx` denotes the number of the current release. An incremental update is also available for updating the previous release `yy` to the current release `xx`. Take the files:

1. README_BEFORE_UNTARRING
2. README_UPDATE
3. untar_me_first.xx.tar.Z
4. update.yy.to.xx.tar.Z
5. update.list

For readers who are not familiar with anonymous ftp, an annotated transaction listing follows in section 2.1.

2.1 Obtaining files by anonymous ftp

Type:		
% ftp 138.67.12.63	—	138.67.12.63 is our ftp site
username: anonymous	—	your username is "anonymous"
password: yourname@your.machine.name	—	type anything here
ftp>	—	this is the prompt you see when you are in ftp

You are now logged in via ftp to the CWP anonymous ftp site. You may type:

ftp> ls	—	to see the contents of the directories
ftp> cd dirname	—	to change directories to "dirname"
ftp> binary	—	to set "binary mode" for transferring files You must do this before you try to transfer any binary file. This includes all files with the form some_name.tar.Z extension.
ftp> get filename	—	to transfer "filename" from our site to your machine
ftp> mget pattern*	—	to transfer all files with names of the "pattern*"
For example:		
ftp> mget *.tar.Z	—	will transfer all files with the form of name.tar.Z to your machine. You will be asked whether you really want each file of this name pattern transferred, before ftp actually does it.
ftp> bye	—	to exit from ftp

2.2 Requirements for installing the package

The only requirements for installing the package are:

1. A machine running the UNIX operating system.
2. Ten megabytes of disk space for the source and compiled binary.

The package has been successfully installed on:

- IBM RS6000
- SUN SPARC STATIONS
- HP 9000 series machines
- HP Apollo
- NeXT
- Convex
- DEC
- Silicon Graphics
- PC's running LINUX, PRIME TIME, ESIX, and NeXTSTEP 486

There are README files in the distribution with special notes about some of these platforms. We depend on the SU user community to alert us to installation problems, so if you run into difficulties, please let us know.

The distribution contains a series of files that detail the installation process. Read them in the following order:

```
LICENSE --- legal statement
README_BEFORE_UNTARRING --- initial information
README_FIRST --- general information
README_TO_INSTALL --- installation instructions
README_X --- X-windows install information
Portability/README_*    --- portability information for various platforms
README_GETTING_STARTED --- how to begin using the codes
```

2.3 A quick test

Once you have completed the installation, here is a quick test you can make to see if you have a functioning seismic system. For an X-windows machine, the "pipeline"

```
suplane | suxwigb &
```

should produce the graphic shown in Figure 2.1. If you have a PostScript printer, then you should get a hard copy version with the pipeline

```
suplane | supswigb | lpr
```

If you have display PostScript, then to get a screen display, replace the `lpr` command in the pipeline by the command that opens a PostScript file.

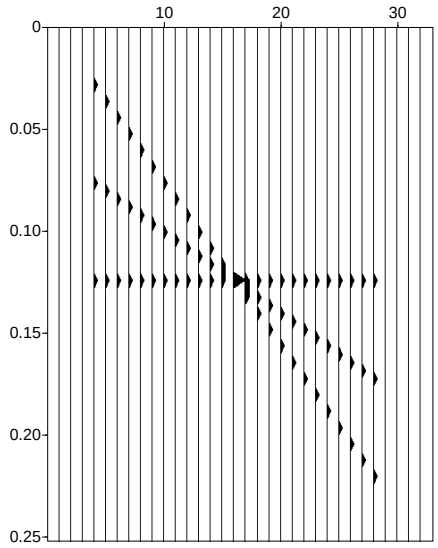

Figure 2.1: Output of the `suplane` pipeline.

Chapter 3

Help Facilities

3.1 self-documentation (selfdoc)

Virtually all SU programs give information about themselves when you type the name of the program without any arguments. For example,

```
% sustack

 SUSTACK - stack adjacent traces having the same key header word

 sustack <input >output key=cdp normpow=1.0 verbose=0

 Required parameters:
        none

 Optional parameters:
        key=cdp         header key word to stack on
        normpow=1.0     each sample is divided by the
                        normpow'th number of non-zero values
                        stacked (normpow=0 selects no division)
        verbose=0       verbose = 1 echos information

 Note:  The offset field is set to zero on the output traces.
        Sushw can be used afterwards if this is not acceptable.
```

3.2 suhelp

suhelp lists the names of the programs in the distribution. The first "page" of the output looks something like this:

```
% suhelp


CWP PROGRAMS: (no self-documentation)
ctrlstrip       fcat            maxints         t
downfort        isatty          pause           upfort
```

```
PAR PROGRAMS: (programs with self-documentation)
a2b             kaperture       resamp          transp          vtlvz
b2a             makevel         smooth2         unif2           wkbj
farith          mkparfile       smoothint2      unisam
ftnstrip        prplot          subset          unisam2
h2b             recast          swapbytes       velconv


press return key to continue
```

3.3 suname

suname is a program that lists the names and a one-line description of all the programs and libraries. Here is the first "page" of the output:

```
% suname

 -----  CWP Free Programs -----

Mains:

In CWPROOT/cwp/main:
* CTRLSTRIP - Strip non-graphic characters
* DOWNFORT - change Fortran programs to lower case, preserving strings
* FCAT - fast cat with 1 read per file
* ISATTY - pass on return from isatty(2)
* MAXINTS - Compute maximum and minimum sizes for integer types
* PAUSE - prompt and wait for user signal to continue
* T - time and date for non-military types
* UPFORT - change Fortran programs to upper case, preserving strings

In CWPROOT/par/main:
A2B - convert ascii floats to binary
B2A - convert binary floats to ascii
FARITH - File ARITHmetic -- perform simple arithmetic with binary files
FTNSTRIP - convert a file of floats plus record delimiters created
H2B - convert 8 bit hexidecimal floats to binary
KAPERTURE - generate the k domain of a line scatterer for a seismic array
MAKEVEL - MAKE a VELocity function v(x,y,z)
MKPARFILE - convert ascii to par file format
--More--(5%)
```

Note that the directory paths are shown as an aid to the SU programmer.

3.4 sudoc

sudoc is a utility that lists the self-documentation of all programs in the package. Subroutines and the few mains and shell scripts that do not show a selfdoc will have an sudoc entry.

For example,

```
% sudoc fgettr

In CWPROOT/src/su/lib:
FGETTR - Routines to get an SU trace from a file

fgettr          get a segy trace from a file by file pointer
gettr           macro using fgettr to get a trace from stdin

Function Prototype:
int fgettr(FILE *fp, segy *tp);

Returns:
int: number of bytes read on current trace (0 after last trace)

The function gettr(x) is a macro defined in su.h
define gettr(x) fgettr(stdin, (x))

Usage example:
        segy tr;
        ...
        while (gettr(&tr)) {
                tr.offset = abs(tr.offset);
                puttr(&tr);
        }
        ...

Authors: SEP: Einar Kjartansson, Stew Levin CWP: Shuki Ronen, Jack Cohen
```

3.5 sufind

sufind is a program that searches the self-documentations for a given string. For example,

```
% sufind fft

 FFTLAB - Motif-X based graphical 1D Fourier Transform

 Usage:  fftlab


HANKEL - Functions to compute discrete Hankel transforms

hankelalloc     allocate and return a pointer to a Hankel transformer
hankelfree      free a Hankel transformer

PFAFFT - Functions to perform Prime Factor (PFA) FFT's, in place

npfa            return valid n for complex-to-complex PFA
npfar           return valid n for real-to-complex/complex-to-real PFA

 SUAMP - output amp, phase, real or imag trace from
        (frequency, x) domain data
```

```
 suamp <stdin >stdout mode=amp

 SUFFT - fft real time traces to complex frequency traces

 suftt <stdin >sdout sign=1


 SUFRAC -- take general (fractional) time derivative or integral of
           data, plus a phase shift.  Input is TIME DOMAIN data.

 sufrac power= [optional parameters] <indata >outdata

 SUIFFT - fft complex frequency traces to real time traces

 suiftt <stdin >sdout sign=-1


 SUMIGPS - MIGration by Phase Shift with turning rays

 sumigps <stdin >stdout [optional parms]


 SUMIGTK - MIGration via T-K domain method for common-midpoint stacked data

 sumigtk <stdin >stdout dxcdp= [optional parms]


 SURADON - forward generalized Radon transform from (x,t) -> (p,tau) space.

 suradon <stdin >stdout [Optional Parameters]



For more information type: "program_name <CR>"
```

The final line of this output ends with a symbol meant to indicate that the user is to type a carriage return.[1]

3.6 sukeyword

SU programs that manipulate the trace headers use specific names called "keywords" to identify the header fields. The **sukeyword** program enables the user to list the definition file for the keywords. For example,

```
% sukeyword fldr

...skipping
        int tracr;      /* trace sequence number within reel */
```

[1] The phrase "carriage return" refers to an older technology, the typewriter. Ask your parents for further details.

```
        int fldr;       /* field record number */

        int tracf;      /* trace number within field record */

        int ep;         /* energy source point number */

        int cdp;        /* CDP ensemble number */

        int cdpt;       /* trace number within CDP ensemble */

        short trid;      /* trace identification code:
                         1 = seismic data
                         2 = dead
                         3 = dummy
                         4 = time break
                         5 = uphole
                         6 = sweep
                         7 = timing
                         8 = water break
                         9---, N = optional use (N = 32,767)
--More--(13%)
```

3.7 Other help mechanisms

- **gendocs** is a program that creates the LaTeX document, **selfdocs.tex**, that contains a complete set of all the self-documentations in the distribution (over 300 pages!). A PostScript version of this document is available in our anonymous ftp site (`pub/cwpcodes/documentation.xx.tar.Z`).

  Here, `xx` denotes the number of the current release.

- The **demos** directory contains a number of tutorial shell scripts. Its subdirectories contain `README` files that give detailed information. Assuming that you start in the **demos** directory, here is a roadmap to get you started:

  The Sorting_Traces Tutorial is an interactive script that reinforces some of the basic UNIX and SU lore discussed in this document. The interactivity is limited to allowing you to set the pace. Such tutorials quickly get annoying, but we felt that one such was needed. The Sorting_Traces Demo is the format used in the rest of the **demos** directory—it just pops up a bunch of windows and assumes you'll read the script if you want to see how a particular window is evoked.

  The next step is to activate the Selecting_Traces Demo. Then proceed to the Deconvolution Demo and the NMO Demo. Beyond that, visit the Demo directories that interest you. The **demos** directory tree is still under active development—please let us know if the demos are helpful and how they can be improved.

- The essence of SU usage is the construction of shell programs to carry out coordinated data processing. The **su/examples** directory contains a number of such programs. By

the way, the terms "shell scripts," "shell programs," "shell files," and "shells," are used interchangeably in the UNIX literature.

- The text book, *Theory of Seismic Imaging*, by John A. Scales, Samizdat Press, 1994, is available in our anonymous ftp site in both 300 and 400 dots per inch PostScript format: `pub/samizdat/texts/imaging/imaging_300dpi.ps.Z` or `imaging_400dpi.ps.Z`. The exercises in this text make extensive use of SU.

- You should not hesitate to look at the source code itself. Section 6.1 explains the key SU coding idioms. Please let us know if you discover any inconsistencies between the source and our documentation of it. We also welcome suggestions for improving the comments and style of our codes.

Chapter 4

Using SU

4.1 SU and UNIX

You need not learn a special seismic language to use SU. If you know how to use UNIX shell-redirecting and pipes, you are ready to start using SU—the seismic commands and options can be used just as you would use the built-in UNIX commands. In particular, you can write ordinary UNIX shell scripts to combine frequent command combinations into meta-commands (i.e., processing flows). These scripts can be thought of as "job files."

Table 4.1: UNIX Symbols

process1 < file1	process1 takes input from file1
process2 > file2	process2 writes on (new) file2
process3 >> file3	process3 appends to file3
process4 | process5	output of process4 is input to process5
process6 << text	take input from following lines

So let's begin with a capsule review of the basic UNIX operators as summarized in Table 4.1. The symbols <, >, and >> are known as "redirection operators," since they redirect input and output into or out of the command (i.e., process). The symbol | is called a "pipe," since we can picture data flowing from one process to another through the "pipe." Here is a simple SU "pipeline" with input "indata" and output "outdata":

```
sufilter f=4,8,42,54 <indata |
sugain tpow=2.0 >outdata
```

This example shows a band-limiting operation being "piped" into a gaining operation. The input data set `indata` is directed into the program **sufilter** with the `<` operator, and similarly, the output data set `outdata` receives the data because of the `>` operator. The output of **sufilter** is connected to the input of **sugain** by use of the `|` operator.

The strings with the `=` signs illustrate how parameters are passed to SU programs. The program **sugain** receives the assigned value 2.0 to its parameter `tpow`, while the program **sufilter** receives the assigned four component *vector* to its parameter `f`. To find out what the valid parameters are for a given program, we use the self-doc facility.

By the way, space around the UNIX redirection and pipe symbols is optional—the example shows one popular style. On the other hand, spaces around the `=` operator are *not* permitted.

The first four symbols in Table 4.1 are the basic grammar of UNIX; the final << entry is the symbol for the less commonly used "here document" redirection. Despite its rarity in interactive use, SU shell programs are significantly enhanced by appropriate use of the << operator—we will illustrate this below.

Many built-in UNIX commands do not have a self-documentation facility like SU's—instead, most do have "man" pages. For example,

```
% man cat

CAT(1)              UNIX Programmer's Manual               CAT(1)



NAME
     cat - catenate and print

SYNOPSIS
     cat [ -u ] [ -n ] [ -s ] [ -v ] file ...

DESCRIPTION
     Cat reads each file in sequence and displays it on the stan-
     dard output.  Thus

                    cat file

     displays the file on the standard output, and

                    cat file1 file2 >file3
--More--
```

You need to know a bit more UNIX lore to use SU efficiently—we'll introduce these tricks of the trade in the context of the examples discussed later in this chapter.

4.2 Exploring SU

This section is a simulated example of an interactive session with SU.

4.2.1 Looking for DMO programs

Later we will discuss the construction of SU processing streams, our present purpose is just to illustrate how to look around for the raw materials for such streams. Let's by using **sufind** to see if there are any DMO programs:

```
% sufind dmo

 SUDMOFK - DMO via F-K domain (log-stretch) method for common-offset gathers

 sudmofk <stdin >stdout cdpmin= cdpmax= dxcdp= noffmix= [...]


 SUDMOTX - DMO via T-X domain (Kirchhoff) method for common-offset gathers
```

```
sudmotx <stdin >stdout cdpmin= cdpmax= dxcdp= noffmix= [optional parms]


SUFDMOD2 - Finite-Difference MODeling (2nd order) for acoustic wave equation

sufdmod2 <vfile >wfile nx= nz= tmax= xs= zs= [optional parameters]


SUSTOLT - Stolt migration for stacked data or common-offset gathers

sustolt <stdin >stdout cdpmin= cdpmax= dxcdp= noffmix= [...]
```

The last two "hits" are spurious, but we see that two DMO programs have been found.

4.2.2 Getting information about SU programs

Use the self-doc facility to get more information about **sudmofk**:

```
% sudmofk

SUDMOFK - DMO via F-K domain (log-stretch) method for common-offset gathers

sudmofk <stdin >stdout cdpmin= cdpmax= dxcdp= noffmix= [...]

Required Parameters:
cdpmin                  minimum cdp (integer number) for which to apply DMO
cdpmax                  maximum cdp (integer number) for which to apply DMO
dxcdp                   distance between adjacent cdp bins (m)
noffmix                 number of offsets to mix (see notes)

Optional Parameters:
tdmo=0.0                times corresponding to rms velocities in vdmo (s)
vdmo=1500.0             rms velocities corresponding to times in tdmo (m/s)
sdmo=1.0                DMO stretch factor; try 0.6 for typical v(z)
fmax=0.5/dt             maximum frequency in input traces (Hz)
verbose=0               =1 for diagnostic print

Notes:
Input traces should be sorted into common-offset gathers.  One common-
offset gather ends and another begins when the offset field of the trace
headers changes.

The cdp field of the input trace headers must be the cdp bin NUMBER, NOT
the cdp location expressed in units of meters or feet.

The number of offsets to mix (noffmix) should typically equal the ratio of
the shotpoint spacing to the cdp spacing.  This choice ensures that every
cdp will be represented in each offset mix.  Traces in each mix will
contribute through DMO to other traces in adjacent cdps within that mix.

The tdmo and vdmo arrays specify a velocity function of time that is
used to implement a first-order correction for depth-variable velocity.
```

```
The times in tdmo must be monotonically increasing.

For each offset, the minimum time at which a non-zero sample exists is
used to determine a mute time.  Output samples for times earlier than this
mute time will be zeroed.  Computation time may be significantly reduced
if the input traces are zeroed (muted) for early times at large offsets.

Trace header fields accessed:  ns, dt, delrt, offset, cdp.
```

4.2.3 Viewing header field definitions

Note that the end of the last program description referred to "header fields"; these names are *not* standard and, as mentioned earlier, you can get definitions by using **sukeyword**. For example,

```
% sukeyword delrt

...skipping
                           may be positive or negative */

        short delrt;    /* delay recording time, time in ms between
                           initiation time of energy source and time
                           when recording of data samples begins
                           (for deep water work if recording does not
                           start at zero time) */

        short muts;     /* mute time--start */

        short mute;     /* mute time--end */

        unsigned short ns;      /* number of samples in this trace */

        unsigned short dt;      /* sample interval; in micro-seconds */

        short gain;     /* gain type of field instruments code:
                                1 = fixed
                                2 = binary
                                3 = floating point
                                4 ---- N = optional use */

--More--(53%)
```

4.2.4 Viewing program names

SU program names are often obscure (we aren't proud of this). Here's how to get help with remembering the exact name of a program when you recall a fragment of the name:

```
% sufind -n head

SUADDHEAD - put headers on bare traces and set the tracl and ns fields
UPDATEHEAD - update ../doc/Headers/Headers.all
```

```
For more information type: "program_name <CR>"
```

Recall also that **suhelp** and **suname** give comprehensive listings of the SU programs.

Note that we used the -n option of the **sufind** command. Using the self-doc facility, we can get the full story:

```
% sufind

sufind - get info from self-docs about SU programs
Usage: sufind [-v -n] string
"sufind string" gives brief synopses
"sufind -v string" verbose hunt for relevant items
"sufind -n name_fragment" searches for command name
```

4.3 Understanding and using SU shell programs

The essence of good SU usage is constructing (or cloning!) UNIX shell programs to create and record processing flows. In this section, we give some annotated examples to get you started.

4.3.1 A simple SU processing flow example

Most SU programs read from standard input and write to standard output. Therefore, one can build complex processing flows by simply connecting SU programs with UNIX pipes. Most flows will end with one of the SU plotting programs. Because typical processing flows are lengthy and involve many parameter settings, it is convenient to put the SU commands in a shell file.

Remark: All the UNIX shells, Bourne (sh), Cshell (csh), Korn (ksh), ..., include a programming language. In this document, we exclusively use the Bourne shell programming language.

Our first example is a simple shell program called **Plot**. The numbers in square brackets at the end of the lines in the following listing are not part of the shell program—we added them as keys to the discussion that follows the listing.

```
#! /bin/sh                                                    [1]
# Plot:   Plot a range of cmp gathers
# Author: Jane Doe
# Usage:  Plot cdpmin cdpmax

data=$HOME/data/cmgs                                          [2]

# Plot the cmp gather.
suwind <$data key=cdp min=$1 max=$2 |                         [3]
sugain tpow=2 gpow=.5 |
suximage f2=0 d2=1 \                                          [4]
        label1="Time (sec)" label2="Trace number" \
        title="CMP Gathers $1 to $2" \
        perc=99 grid1=solid &                                 [5]
```

Discussion of numbered lines:

1. The symbol `#` is the comment symbol—anything on the remainder of the line is not executed by the UNIX shell. The combination `#!` is an exception to this rule: the shell uses the file name following this symbol as a path to the program that is to execute the remainder of the shell program.

2. The author apparently intends that the shell be edited if it is necessary to change the data set—she made this easier to do by introducing the shell variable `data` and assigning to it the full pathname of the data file. The assigned value of this parameter is accessed as `$data` within the shell program. The parameter `$HOME` appearing as the first component of the file path name is a UNIX maintained environment variable containing the path of the user's home directory. In general, there is no need for the data to be located in the user's home directory, but the user would need "read permission" on the data file for the shell program to succeed.

   WARNING! Spaces are significant to the UNIX shell—it uses them to parse command lines. So despite all we've learned about making code easy to read, do *not* put spaces next to the `=` symbol. (Somewhere around 1977, one author's (Jack) first attempt to learn UNIX was derailed for several weeks by making this mistake.)

3. The main pipeline of this shell code selects a certain set of cmp gathers with **suwind**, gains this subset with **sugain** and pipes the result into the plotting program **suximage**. As indicated in the Usage comment, the cmp range is specified by command line arguments. Within the shell program, these arguments are referenced as `$1`, `$2` (i.e., first argument, second argument).

4. The lines within the **suximage** command are continued by the backslash escape character.

   WARNING! The line continuation backslash must be the *final* character on the line—an invisible space or tab following the backslash is one of the most common and frustrating bugs in UNIX shell programming.

5. The final `&` in the shell program puts the plot window into "background" so we can continue working in our main window. This is the X-Windows usage—the `&` should *not* be used with the analogous PostScript plotting programs (e.g., supsimage). For example, with **supsimage** in place of **suximage**, the `&` might be replaced by `| lpr`.

   The SU plotting programs are special—their self-doc doesn't show all the parameters accepted. For example, most of the parameters accepted by **suximage** are actually specified in the self-documentation for the generic CWP plotting program **ximage**. This apparent flaw in the self-documentation is actually a side effect of a key SU design decision. The SU graphics programs call on the generic plotting programs to do the actual plotting. The alternative design was to have tuned graphics programs for various seismic applications. Our design choice keeps things simple, but it implies a basic limitation in SU's graphical capabilities.

   The plotting programs are the vehicle for presenting your results. Therefore you should take the time to carefully look through the self-documentation for *both* the "SU jacket" programs (**suximage**, **suxwigb**, ...) and the generic plotting programs (**ximage**, **xwigb**, ...).

4.3.2 Executing shell programs

The simplest way to execute a UNIX shell program is to give it "execute permission." For example, to make our above **Plot** shell program executable:

```
chmod +x Plot
```

Then to execute the shell program:

```
Plot 601 610
```

Here we assume that the parameters `cdpmin=601`, `cdpmax=610` are appropriate values for the `cmgs` data set. Figure 4.1 shows an output generated by the `Plot` shell program.

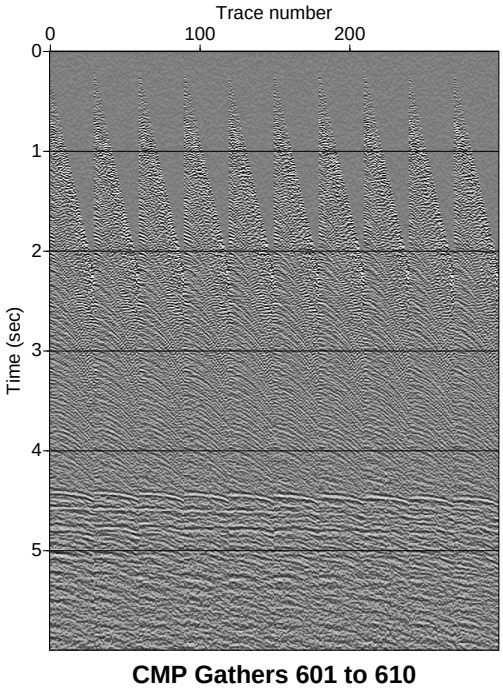

Figure 4.1: Output of the `Plot` shell program.

4.3.3 A typical SU processing flow

Suppose you want to use **sudmofk**. You've read the self-doc, but a detailed example is always welcome isn't it? The place to look is the directory **su/examples**. In this case, we are lucky and find the shell program, **Dmo**. Again, the numbers in square brackets at the end of the lines shown below are *not* part of the listing.

```
#! /bin/sh
# dmo
set -x                                                          [1]

# set parameters
input=cdp201to800                                               [2]
```

```
temp=dmocogs
output=dmocmgs
smute=1.7
vnmo=1500,1550,1700,2000,2300,2600,3000                              [3]
tnmo=0.00,0.40,1.00,2.00,3.00,4.00,6.00


# sort to common-offset, nmo, dmo, inverse-nmo, sort back to cmp
susort <$input offset cdp |                                          [4]
sunmo smute=$smute vnmo=$vnmo tnmo=$tnmo |                           [5]
sudmofk cdpmin=201 cdpmax=800 dxcdp=13.335 noffmix=4 verbose=1 |     [6]
sunmo invert=1 smute=$smute vnmo=$vnmo tnmo=$tnmo >$temp             [7]
susort <$temp cdp offset >$output                                    [8]
```

Discussion of numbered lines:

The core of the shell program (lines 5-7) is recognized as the typical dmo process: crude nmo, dmo, and then "inverse" nmo. The dmo processing is surrounded by sorting operations (lines 4 and 8). Here is a detailed discussion of the shell program keyed to the numbers appended to the listing (see also the discussion above for the `Plot` shell):

1. Set a debugging mode that asks UNIX to echo the lines that are executed. You can comment this line off when its output is no longer of interest. An alternate debugging flag is `set -v` which echos lines as they are read by the shell interpreter. You can use both modes at once if you like.

2. This line and the next two lines set filenames that, in this case, are in the same directory as the shell program itself. Again, the reason for using parameters here is to make it easy to "clone" the shell for use with other data sets. Those of us who work with only a few data sets at any given time, find it convenient to devote a directory to a given data set and keep the shells used to process the data in that directory as documentation of the processing parameters used. (SU does not have a built-in "history" mechanism.)

3. The dmo process requires a set of velocity-time picks for the subsidiary nmo processes. Since these picks must be consistent between the nmo and the inverse nmo, it is a good idea to make them parameters to avoid editing mistakes. Again, note the format of SU parameter vectors: comma-separated strings with no spaces. The nmo program (**sunmo**) will give an error message and abort if the `vnmo` and `tnmo` vectors have different lengths.

4. Note that **susort** allows the use of *secondary* sort keys. Do not assume that a secondary field that is initially in the "right" order will remain in that order after the sort—if you care about the order of some secondary field, specify it (as this shell program does). In this line, we sort the data according to increasing offsets and then, within each offset, we sort according to increasing cdp number.

5. The forward nmo step.

6. The dmo step.

7. The inverse nmo step.

8. Sort back to cdp and have increasing offset within each cdp.

If you want to thoroughly understand this shell program, your next step is to study the self-docs of the programs involved:

```
% sunmo

SUNMO - NMO for an arbitrary velocity function of time and CDP

sunmo <stdin >stdout [optional parameters]

Optional Parameters:
vnmo=2000         NMO velocities corresponding to times in tnmo
tnmo=0            NMO times corresponding to velocities in vnmo

...
```

Related shell programs are **su/examples/Nmostack** and **su/examples/Mig**.

4.4 Extending SU by shell programming

Shell programming can be used to greatly extend the reach of SU without writing C code. See, for example, **Cvstack**, **Filtertest**, **Firstbreak**, and **Velan** in **/su/examples**.

It is a sad fact that the UNIX shell is not a high level programming language—consequently effective shell coding often involves arcane tricks. In this section, we'll provide some useful templates for some of the common UNIX shell programming idioms.

We use **Cvstack** as an illustration. The core of this shell is a double loop over velocities and cdps that produces *velocity panels*—a concept not contained in any single SU program.

Remark: For most of us, writing a shell like **Cvstack** from scratch is a time-consuming affair. To cut down the development time, your authors excerpt from existing shells to make new ones even when we don't quite remember what every detail means. We suggest that you do the same!

We won't comment on the lines already explained in our previous two shell code examples (see Sections 4.3.1 and 4.3.3), but instead focus on the new features used in **Cvstack**.

```
#! /bin/sh
# Constant-velocity stack of a range of cmp gathers
# Authors: Jack, Ken
# NOTE: Comment lines preceding user input start with  #!#
set -x

#!# Set input/output file names and data parameters
input=cdp601to610
stackdata=cvstack
cdpmin=601 cdpmax=610
fold=30
space=1         # 1 null trace between panels

#!# Determine velocity sampling.
vmin=1500   vmax=3000   dv=150

### Determine ns and dt from data (for sunull)
```

```
nt=`sugethw ns <$input | sed 1q | sed 's/.*ns=//'`                [1]
dt=`sugethw dt <$input | sed 1q | sed 's/.*dt=//'`

### Convert dt to seconds from header value in microseconds
dt=`bc -l <<END                                                  [2]
        scale=4
        $dt / 1000000
END`


### Do the velocity analyses.
>$stackdata  # zero output file                                  [3]
v=$vmin
while [ $v -le $vmax ]                                           [4]
do
        cdp=$cdpmin
        while [ $cdp -le $cdpmax ]                               [5]
        do
                suwind <$input \                                 [6]
                        key=cdp min=$cdp max=$cdp count=$fold |
                sunmo cdp=$cdp vnmo=$v tnmo=0.0 |
                sustack >>$stackdata
                cdp=`bc -l <<END                                 [7]
                        $cdp + 1
END`
        done
        sunull ntr=$space nt=$nt dt=$dt >>$stackdata             [8]
v=`bc -l <<END
         $v + $dv
END`
done


### Plot the common velocity stacked data
ncdp=`bc -l <<END
        $cdpmax-$cdpmin+1
END`
f2=$vmin
d2=`bc -l <<END
        $dv/($ncdp + $space)                                     [9]
END`

sugain <$stackdata tpow=2.0 |

suximage perc=99 f2=$f2 d2=$d2 \
        title="File: $input  Constant-Velocity Stack " \
        label1="Time (s)"  label2="Velocity (m/s)" &

exit                                                             [10]
```

Discussion of numbered lines:

1. This elaborate construction gets some information from the first trace header of the data

set. The program **sugethw** lists the values of the specified keys in the successive traces. For example,

```
% suplane | sugethw tracl ns
 tracl=1            ns=64

 tracl=2            ns=64

 tracl=3            ns=64

 tracl=4            ns=64

 tracl=5            ns=64

 tracl=6            ns=64

 ...
```

Although **sugethw** is eager to give the values for every trace in the data set, we only need it once. The solution is to use the UNIX stream editor (**sed**). In fact, we use it twice. By default, **sed** passes along its input to its output. Our first use is merely to tell **sed** to quit after it puts the first line in the pipe. The second pass through **sed** strips off the unwanted material before the integer. In detail, the second **sed** command reads: replace (or substitute) everything up to the characters `ns=` with nothing, i.e., delete those characters.

2. We are proud of this trick. The Bourne shell does not provide floating point arithmetic. Where this is needed, we use the UNIX built-in **bc** calculator program with the "here document" facility. Here, we make the commonly needed conversion of sampling interval which is given in micro-seconds in the SEG-Y header, but as seconds in SU codes. Note carefully the *back*quotes around the entire calculation—we assign the result of this calculation to the shell variable on the left of the equal sign, here `dt`. The calculation may take several lines. We first set the number of decimal places with `scale=4` and then do the conversion to seconds. The characters `END` that follow the here document redirection symbol `<<` are arbitrary, the shell takes its input from the text in the shell file until it comes to a line that contains the same characters again. For more information about **bc**:

   ```
   % man bc
   ```

3. As the comment indicates, this is a special use of the output redirection symbol that has the effect of destroying any pre-existing file of the same name or opening a new file with that name. In fact, this is what `>` always does as its first action—it's a dangerous operator! If you intend to *append*, then, as mentioned earlier, use `>>`.

4. This is the outer loop over velocities. Another warning about spaces—the spaces around the bracket symbols are essential.

   Caveat: The bracket notation is a nice alternative to the older clunky `test` notation:

   ```
   while test $v -le $vmax
   ```

Since the bracket notation is not documented on the typical **sh** manual page, we have some qualms about using it. But, as far as we know, all modern **sh** commands support it—please let us know if you find one that doesn't.

WARNING! OK, now you know that there is a UNIX command called `test`. So don't use the name "test" for one of your shell (or C) programs—depending on your `$PATH` setting, you could be faced with seemingly inexplicable output.

5. This is the inner loop over cdps.

6. Reminder: No spaces or tabs after the line continuation symbol!

7. Notice that we broke the nice indentation structure by putting the final `END` against the left margin. That's because the **sh** manual page says that the termination should contain only the `END` (or whatever you use). In fact, most versions support indentation. We didn't think the added beautification was worth the risk in a shell meant for export. Also note that we used **bc** for an integer arithmetic calculation even though integer arithmetic is built into the Bourne shell—why learn two arcane rituals, when one will do? See `man expr`, if you are curious.

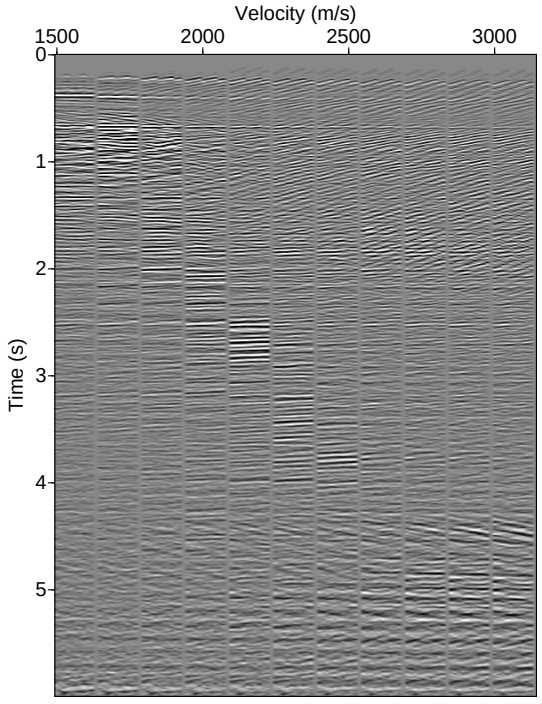

Figure 4.2: Output of the `Cvstack` shell program.

8. **sunull** is a program I (Jack) wrote to create all-zero traces to enhance displays of the sort produced by `Cvstack`. Actually, I had written this program many times, but this was the first time I did it on purpose. (Yes, that was an attempt at humor.)

9. An arcane calculation to get velocity labeling on the trace axis. Very impressive! I wonder what it means? (See last item.)

10. The `exit` statement is useful because you might want to save some "spare parts" for future use. If so, just put them after the `exit` statement and they won't be executed.

Figure 4.2 shows an output generated by `Cvstack`.

4.5 Some core programs

Reading the self-documentation and trying out the following SU programs will give you a good start in learning SU.

4.5.1 Examining the trace headers

surange — print minimum and maximum values of trace header fields

sugethw — print values of selected header fields

suascii — print header and data values

suxedit — interactively examine headers and traces

4.5.2 Some common processing programs

suacor — compute autocorrelations

sufilter — multipurpose zero phase filter (includes bandpass)

sugain — gain (with lots of options)

sumute — zero samples before a time that depends on offset

sunmo — normal-moveout correction

supef — prediction error filtering

susort — sort traces by values of trace header fields

sustack — stack (sum) traces

suvelan — velocity analysis

suwind — window (i.e., get a subset of) traces

4.5.3 Some common plotting programs

suximage — gray scale X Windows plotting

suxwigb — bit mapped wiggle trace X Windows plotting

supsimage — gray scale PostScript plotting

supswigb — bit mapped wiggle trace PostScript plotting

4.6 A brief tour of the source directories

The SU software is a layered product. The layers correspond to the following directories:

cwp Library of scientific routines (e.g. fft routines) written in "vanilla" C. Utility mains and shells.

par Library supporting the CWP programming style (i.e., self-doc, error reporting, parameter passing). Mains that use (only) these facilities. Shells for maintaining the online documentation database.

su Seismic processing codes that use the SEG-Y trace structure. Subroutines that manage this structure. Codes that buffer the generic graphics routines listed below. Shells that provide backward compatibility with earlier releases.

graphics libraries

1. psplot—PostScript graphics:
   (a) pscontour: contour plots
   (b) pscube: 3D data cube
   (c) psgraph: curve plotting
   (d) psimage: raster plotting
   (e) psmovie: supports frames
   (f) pswigb: bit mapped wiggle traces (fast)
   (g) pswigp: polygon wiggle traces (slow)
   (h) PostScript support programs
2. xplot—xlib based X Windows graphics
   (a) ximage: raster plotting
   (b) xwigb: bit mapped wiggle traces
   (c) X Windows support programs
3. Xtcwp—toolkit based X Windows graphics
   (a) xgraph: curve plotting
   (b) xmovie: supports frames
   (c) X Windows resource files

These are only the highlights. If you intend to add your own C mains to the package, it is worthwhile spending a few hours browsing through the source code.

Chapter 5

Frequently Asked Questions

This chapter addresses questions often asked by new SU users. Some answers refer to the directory CWPROOT. We use this symbolic name for the directory that contains the CWP/SU source code, include files, libraries, and executables. You are asked to specify this directory name during the SU installation procedure.

5.1 Installation questions

Complete information about the installation process is found in the README files supplied with the distribution. Here we discuss only some commonly found installation problems.

Question 1 *I get error messages about missing* `fgetpos` *and* `fsetpos` *routines, even though I am using the* GCC *compiler. How do I get around this problem?*

Answer 1 We've seen this problem most often with older SUN OS 4.xx (pre-SOLARIS). These SUN systems may not have the `fgetpos` and `fsetpos` subroutines defined. Since these two routines are not currently used in the SU package, the easiest fix when these two functions are missing functions is to comment out references to the functions `efgetpos` and `efsetpos` in `CWPROOT/src/par/include/par.h` and `CWPROOT/src/par/lib/subcalls.c`. See also the answer to Question 3.

Question 2 *I get error messages regarding missing* `strtoul`*, and/or* `strerror` *routines, even though I am using the* GCC *compiler. How do I get around this problem?*

Answer 2 Again, this is most often seen with the older SUN OS. To fix the problem of a missing `strtoul` routine, replace:
`CWPROOT/src/par/lib/atopkge.c`
with the file:
`CWPROOT/src/par/lib/Portability/atopkge.c`.

For a missing `strerror` routine, replace:
`CWPROOT/src/par/lib/errpkge.c`
with the file:
`CWPROOT/src/par/lib/Portability/errpkge.c`.

Each of these replacements, of the regular file by the `Portability` version, entails a small loss of functionality. See also the answer to Question 3.

Question 3 *Why do I get missing subroutine messages about* ANSI C *routines? Isn't the* GCC *compiler supposed to be an* ANSI *compiler?*

Answer 3 The GCC compiler is just that, a compiler. It draws on the libraries that are present on the machine. If the GNU libraries have not been installed, then the GCC compiler will use the libraries that are native to the machine you are running on. Because the four routines listed above are not available in the SUN 4. OS, GCC does not recognize them. However, installing the GNU libraries will make the GCC compiler behave as a full ANSI C compiler.

5.2 Data format questions

In this section, we address questions about converting data that are in various formats into SU format.

Question 4 *What is the data format that* SU *programs expect?*

Answer 4 The SU data format is based on the SEG-Y format. The SU format consists of data traces each of which has a header. The SU trace header is identical to SEG-Y trace header. Both the header and the trace data are written in the native binary format of your machine.
Caution: The optional fields in the SEG-Y trace header are used for different purposes at different sites. SU itself makes use of certain of these fields. Thus, you may need to use `segyclean`—see the answer to Question 6. SU format does not have the binary and ebcdic tape headers that are part of the SEG-Y format.

After installing the package, you can get more information on the SEG-Y/SU header by typing:

```
% sukeyword -o
```

This lists the include file `segy.h` that defines the SU trace header.

Question 5 *Is there any easy way of adding necessary* SEG-Y *information to our own modeled data to prepare our data for processing using the* SU *package?*

Answer 5 It depends on the details of how your data was written to the file:

1. If you have a 'datafile' that is in the form of binary floating point numbers of the type that would be created by a C program, then use `suaddhead` to put SU (SEG-Y) trace headers on the data. Example:

   ```
   % suaddhead < datafile  ns=N_SAMP > data.su
   ```

   Here, `N_SAMP` is the (integer) number of samples per trace in the data.

2. If your data are Fortran-style floats, then you would use:

   ```
   % suaddhead < datafile ftn=1 ns=NS > data.su
   ```

   See also, Question 9.

3. If your data are ASCII, then use:

```
% a2b n1=N1 < data.ascii | suaddhead ns=NS > data.su
```

Here `N1` is the number of floats per line in the file `data.ascii`.

4. If you have some other data type, then you may use:

```
% recast < data.other in=IN out=float | suaddhead ns=NS > data.su
```

where `IN` is the type (int, double, char, etc...)

For further information, consult the self-docs of the programs `suaddhead`, `a2b`, and `recast`.

Question 6 *I used* `segyread` *to read a* SEG-Y *tape. Everything seems to work fine, but when I plot my data with suximage, the window is black. What did I do wrong?*

Answer 6 When you read an SEG-Y tape, you need to pipe the data through `segyclean` to zero the optional SEG-Y trace header field. If the SU programs see nonzero values in certain parts of the optional field, they try to display the data as "nonseismic data," using those values to set the plot parameters.

Question 7 *I am trying to plot data with the* `pswigb` *(or* `pswigp`*, or* `xwigb`*, or ...) program. I know that I have data with* `n1=NSAMP` *and* `n2=NTRACES`*, but when I plot, I find that I have to set* `n1=NSAMP+60` *for the plot to look even remotely correct. Why is this?*

Answer 7 It is likely that you are trying to plot with the wrong tool. The input data format of the programs, `pswigb`, `pswigp`, `pscontour`, `pscube`, `psmovie`, `xwigb`, `xgraph`, and `xmovie`, expect data to consist of simple floating point numbers. If your data are SU data (SEG-Y) traces, then there is an additional header at the beginning of each trace, which, on most computer architectures, is the same number (240) of bytes as the storage for 60 floats.

To plot these data, use respectively: `supswigb`, `supswigp`, `supscontour`, `supscube`, `supsmovie`, `suxwigb`, `suxgraph`, or `suxmovie`.

Also, it is not necessary to specify the dimensions of the data for these latter programs. The `su`-versions of the codes determine the necessary information from the appropriate header values. (In fact, that is *all* they do—the actual graphics is handled by the version without the `su` prefix.)

Question 8 *I want to check the size of a file to see if it has the right number of values, but I am not sure how to take the header into account. How is this done?*

Answer 8 If the file consists of simple floating point numbers, then the size in bytes equals the size of a float times the number of samples (`SIZE = 4 * N_SAMP`). The SU data (SEG-Y traces) also have a header (240 bytes per trace) giving the total number of bytes as:
`(240 + 4 N_SAMP ) N_TRACES`.
The byte count computed in this way is the number that the UNIX command `ls -l` shows.

Caveats: The above calculations assume that you have the conventional architecture and that the header definition in `segy.h` has not been altered. Watch out as machines with 64 bit word size become common!

Question 9 *I have some data in Fortran form and tried to convert it to* SU *data via the following:*

```
% suaddhead < data.fortran ns=N_SAMP ftn=1 > data.su
```

but this did not work properly. I am sure that my fortran data are in unformatted binary floats. What should I do?

Answer 9 There are different ways of interpreting the term "unformatted" with regard to fortran data. Try:

```
% ftnstrip < data.fortran | suaddhead ns=N_SAMP > data.su
```

The program `ftnstrip` can often succeed in converting your fortran data into C-like binary data, even when the `ftn=1` option in `suaddhead` fails.

Question 10 *I just successfully installed the* CWP/SU *package, but when I try to run the demo scripts, I get many error messages describing programs that the shell script cannot find. How do I fix this?*

Answer 10 You need to put `CWPROOT/bin` (where `CWPROOT` is `/your/root/path` that contains the CWP/SU source code, include files, libraries, and executables) in your shell `PATH`. This is done in your `.cshrc` file if you run under `csh` or `tcsh`. In Bourne shell (`sh`), Born Again shell (`bash`), or Korn shell (`ksh`) the `PATH` variable is in your `.profile` file. You also need to type

```
% rehash
```

under `.csh` or `.tcsh`, if you have not relogged since you compiled the codes.

5.3 Tape reading and writing

This section contains frequently asked questions about reading and writing SEG-Y tapes with SU.

Tape reading/writing is more of an art than a science. Here are a few tips.

1. Make sure your tape drive is set to be variable block length. If you are on an IBM RS6000, this means you will need to use `smit` to set `blocksize=0` on your tape device. Having the tape drive set to some default constant blocksize (say blocksize=1024 or 512) will foil all attempts to read an SEG-Y tape.

2. To read multiple tape files on a tape, use the non rewinding device. On an `RS6000` this would be something like `/dev/rmtx.1`, see `man mt` for details.

3. If this still doesn't work, then try:

   ```
   % dd if=/dev/rmtx of=temps bs=32767 conv=noerror
   ```

   Here, `/dev/rmtx` (not the real name of the device, it varies from system to system) is your regular (rewinding) tape device. In the option, `bs=32767`, we gave the right blocksize $(2^{16}+1)$ for an `IBM/RS6000`. Try `bs=32765` $(2^{16}-1)$ on a `SUN`. This will dump the entire contents of the tape onto a single file.

Question 11 *How do I write multiple SEG-Y files onto a tape?*

Answer 11 Here is a shell script for writing multiple files on a tape:

```
#! /bin/sh

DEV=/dev/nrxt0  # non rewinding tape device

mt -f $DEV rewind

j=0
jmax=40

while test "$j" -ne "$jmax"
do
j=`expr $j + 1`
echo "writing tape file  $j"
segywrite tape=$DEV bfile=b.$j hfile=h.$j verbose=1 buff=0 < ozdata.$j
done

exit 0
```

5.4 General

This section addresses general questions about the SU package.

Question 12 *Why are* CWP/SU *releases given by integers (22, 23, 24, etc...) instead of the more familiar decimal release numbers (1.1, 1.3, etc...)?*

Answer 12 The CWP/SU release numbers are chosen to correspond to the SU NEWS email messages. The individual codes in the package have traditional decimal release numbers (assigned by RCS), but these are all different. The package changes in incremental, but non-uniform ways, so the standard notation seems inappropriate. However, the user may view 24 to be 2.4. We may adopt this convention in the future.

Remark: In the early days, we *did* use RCS to simultaneously update all the codes to 2.1, 3.1, ... This practice died a natural death somewhere along the way.

Question 13 *How often are the codes updated?*

Answer 13 The CWP/SU package is updated at roughly 3 month intervals. We mail announcements of these releases to all known users. Since we do not provide support for outdated versions, we urge you to remain current.

Question 14 *I have a complicated collection of input parameters for a* CWP/SU *program. I want to run the command from the command line of a terminal window, but I don't want to retype the entire string of input parameters. what do I do?*

Answer 14 CWP/SU programs that take their input parameters from the command line also have the feature of being able to read from a "parameter file." This is invoked by setting the parameter `par=parfile`, where `parfile` is a file containing the desired commandline string.

For example:

```
suplane ntr=20 nt=40 dt=.001 | ...
```

is completely equivalent to the command:

```
suplane par=parfile | ...
```

The string

```
ntr=20 nt=40 dt=.001
```

is contained in 'parfile.'

Chapter 6

How to Write an SU Program

6.1 A template SU program

Although variations are usually needed, a template for a typical SU program looks like the program listing below (we excerpted lines from the program `sumute` to build this template). The numbers in square brackets at the end of the lines in the listing are not part of the listing—we added them to facilitate discussion of the template. The secret to efficient SU coding is finding an existing program similar to the one you want to write. If you have trouble locating the right code or codes to "clone," ask us—this can be the toughest part of the job!

```
/* SUMUTE: $Revision: 1.13 $ ; $Date: 93/11/24 15:50:02 $ */  [1]

#include "su.h"                                                [2]
#include "segy.h"

/************************ self documentation **********************/ [3]
char *sdoc[] = {
"                                                                ",
" SUMUTE - ......                                                ",
"                                                                ",
" sumute <stdin >stdout                                          ",
"                                                                ",
" Required parameters:                                           ",
"         none                                                   ",
"                                                                ",
" Optional parameters:                                           ",
"         ...                                                    ",
"                                                                ",
" Trace header fields accessed: ns                               ",
" Trace header fields modified: none                             ",
"                                                                ",
NULL};
/***************** end self doc ***********************************/

/* Credits:
 *
 *        CWP: Jack Cohen, John Stockwell
 */
```

```
segy tr;                                                        [4]

main(int argc, char **argv)
{
        int ns;                 /* number of samples          */      [5]
        ...


        /* Initialize */
        initargs(argc, argv);                                   [6]
        requestdoc(1);                                          [7]

        /* Get parameters */
        if (!getparint("ntaper", &ntaper))        ntaper = 0;   [8]


        /* Get info from first trace */
        if (!gettr(&tr)) err("can't read first trace");         [9]
        if (!tr.dt) err("dt header field must be set");         [10]

        /* Loop over traces */
        do {                                                    [11]
                int nt     = (int) tr.ns;                       [12]

                if (below == 0) {                               [13]
                        nmute = NINT((t - tmin)/dt);
                        memset((void *) tr.data, (int) '\0', nmute*FSIZE);
                        for (i = 0; i < ntaper; ++i)
                                tr.data[i+nmute] *= taper[i];
                } else {
                        nmute = NINT((nt*dt - t)/dt);
                        memset((void *) (tr.data+nt-nmute),
                                        (int) '\0', nmute*FSIZE);
                        for (i = 0; i < ntaper; ++i)
                                tr.data[nt-nmute-1-i] *= taper[i];
                }
                puttr(&tr);                                      [14]
        } while (gettr(&tr));                                    [15]

        return EXIT_SUCCESS;                                     [16]
}
```

Discussion of numbered lines:

1. We maintain the internal versions of the codes with the UNIX utility RCS. This item shows the string template for RCS.

2. The file `su.h` includes (directly or indirectly) all our locally defined macros and prototypes. The file `segy.h` has the definitions for the trace header fields.

3. The starred lines delimit the "self-doc" information—include them exactly as you find them in the codes since they are used by the automatic documentation shells. The style of the self-doc shown is typical except that often additional usage information is shown at the bottom and, of course, often there are more options. Look at some existing codes for ideas.

4. This is an external declaration of an SU (SEG-Y) trace buffer. It is external to avoid wasting stack space.

5. We usually describe the global variables at the time of declaration. Examine codes related to yours to increase consistency of nomenclature (there is no official SU naming standard).

6. The `initargs` subroutine sets SU's command line passing facility (see page 13).

7. The `requestdoc` subroutine call specifies the circumstances under which self-doc will be echoed to the user. The argument '1' applies to the typical program that uses only standard input (i.e. `<`) to read an SU trace file. Use '0' for codes that create synthetic data (like `suplane`) and '2' for codes that require two input files (we could say "et cetera," but there are no existing SU mains that require *three* or more input files).

8. This is typical code for reading 'parameters from the command line. Interpret it like this: "If the user did not specify a value, then use the default value." The subroutine must be type-specific, here we are getting an *integer* parameter.

9. Read the first trace, exit if empty. The subroutine `fgettr` "knows about" the SU trace format. Usually the trace file is read from standard input and then we use `gettr` which is a macro based on `fgettr` defined in `su.h`. Note that this code implies that the first trace is read into the trace buffer (here called `tr`), therefore we will have to process this trace before the next call to `fgettr`.

10. We've read that first trace because, we need to get some trace parameters from the first trace header. Usually these are items like the number of samples (`tr.ns`) and/or the sampling interval (`tr.dt`) that, by the SEGY-Y standard, are the same for all traces.

11. Since the first trace has been (typically) read before the main processing loop starts, we use a "do-while" that reads a new trace at the *bottom* of the loop.

12. We favor using *local* variables where permitted.

13. This is the seismic algorithm–here incomplete. We've left in some of the actual `sumute` code because it happens to contains lines that will be useful in the new code, we'll be writing below. You may want to call a subroutine here to do the real work.

14. `fputtr` and `puttr` are the output analogs of `fgettr` and `gettr`.

15. The loop end. `gettr` returns a 0 when the trace file is exhausted and the processing then stops.

16. This is an ANSI-C macro conventionally used to indicate successful program termination.

6.2 Writing a new program: suvlength

A user asked about SU processing for variable length traces. At his institute, data are collected from time of excitation to a variable termination time. The difficulty is that SU processing is based on the SEG-Y standard which mandates that all traces in the data set be of the same length. Rather than contemplating changing all of SU, it seems to us that the solution is to provide a program that converts the variable length data to fixed length data by padding with zeroes where necessary at the end of the traces—let's name this new program `suvlength`. We can make the length of the output traces a user parameter. If there is a reasonable choice, it makes sense to provide a default value for parameters. Here, using the length of the first trace seems the best choice since that value can be ascertained before the main processing loop starts.

So far, so good. But now our plan runs into a serious snag: the fundamental trace getting facility, `gettr`, itself assumes fixed length traces (or perhaps we should say that `gettr` deliberately enforces the fixed length trace standard). But, if you think about it, you'll realize that `gettr` itself has to take special measures with the *first* trace to figure out its length. All we have to do is make a new trace getting routine that employs that first trace logic for *every* trace! Fortunately, we don't even have to do that since the same problem arose a few years ago and we wrote `fvgettr` at that time for the requirements of the `suvcat` program. So as a first draft solution, we'll just copy in `fvgettr` as an in-code subroutine for our new program. Let's begin converting our above template into the new `suvlength` code:

```
/* SUVLENGTH: $Revision: 1.1 $ ; $Date: 94/07/06 11:19:53 $ */

#include "su.h"
#include "segy.h"

/************************ self documentation **********************/
char *sdoc[] = {
"                                                                  ",
" SUVLENGTH - Adjust variable length traces to common length      ",
"                                                                  ",
" suvlength <variable_length_traces >fixed_length_traces          ",
"                                                                  ",
" Required parameters:                                             ",
"         none                                                     ",
"                                                                  ",
" Optional parameters:                                             ",
"        ns      output number of samples (default: 1st trace ns)",
"                                                                  ",
" Trace header fields accessed: ns                                 ",
" Trace header fields modified: ns                                 ",
"                                                                  ",
NULL};
/***************** end self doc ***********************************/

/* Credits:
 *
 *        CWP: Jack Cohen, John Stockwell
 */
```

```
/* prototype */
int fvgettr(FILE *fp, segy *tp);

segy tr;

main(int argc, char **argv)
{
        int ns;         /* number of samples on output traces  */


        /* Initialize */
        initargs(argc, argv);
        requestdoc(1);

        /* Get parameters */
        ...

        /* Get info from first trace */
        ...

        ...

        return EXIT_SUCCESS;                                          [16]
}

/* fvgettr code from suvcat goes here */
        ...
```

Now we run into a small difficulty. Our only parameter has a default value that is obtained only after we read in the first trace. The obvious solution is to reverse the parameter getting and the trace getting in the template. Thus we resume:

```
        /* Get info from first trace and set ns */
        if (!fvgettr(stdin, &tr))  err("can't get first trace");
        if (!getparint("ns", &ns))    ns = tr.ns;

        /* Loop over the traces */
        do {
                int nt = tr.ns;
```

Now comes the actual seismic algorithm—which is rather trivial in the present case: add zeroes to the end of the input trace if the output length is specified greater than the input length. We could write a simple loop to do the job, but the task is done most succinctly by using the ANSI-C routine `memset`. However, we confess that unless we've used it recently, we usually forget how to use this routine. One solution is to `cd` to the `su/main` directory and use `grep` to find other uses of `memset`. When we did this, we found that `sumute` had usage closest to what we needed and that is why we started from a copy of that code. Here is the complete main for `suvlength`:

```
/* SUVLENGTH: $Revision: 1.1 $ ; $Date: 94/07/06 11:19:53 $        */

#include "su.h"
#include "segy.h"
```

```
/*********************** self documentation **********************/
char *sdoc[] = {
"                                                                 ",
" SUVLENGTH - Adjust variable length traces to common length      ",
"                                                                 ",
" suvlength <vdata >stdout                                        ",
"                                                                 ",
" Required parameters:                                            ",
"         none                                                    ",
"                                                                 ",
" Optional parameters:                                            ",
"          ns     output number of samples (default: 1st trace ns)",
"                                                                 ",
" Trace header fields accessed:  ns                               ",
" Trace header fields modified:  ns                               ",
"                                                                 ",
NULL};
/**************** end self doc ***********************************/

/* Credits:
 *        CWP: Jack Cohen, John Stockwell
 */

/* prototype */
int fvgettr(FILE *fp, segy *tp);

segy tr;

main(int argc, char **argv)
{
        int ns;                /* samples on output traces        */


        /* Initialize */
        initargs(argc, argv);
        requestdoc(1);


        /* Get info from first trace */
        if (!fvgettr(stdin, &tr))  err("can't get first trace");
        if (!getparint("ns", &ns))    ns = tr.ns;


        /* Loop over the traces */
        do {
                int nt = tr.ns;

                if (nt < ns) /* pad with zeros */
                        memset((void *)(tr.data + nt), '\0', (ns-nt)*FSIZE);
                tr.ns = ns;
                puttr(&tr);
```

```
        } while (fvgettr(stdin, &tr));

        return EXIT_SUCCESS;
}


#include "header.h"

/* fvgettr - get a segy trace from a file by file pointer (nt can vary)
 *
 * Returns:
 *        int: number of bytes read on current trace (0 after last trace)
 *
 * Synopsis:
 *        int fvgettr(FILE *fp, segy *tp)
 *
 *
 * Credits:
 *        Cloned from .../su/lib/fgettr.c
 *
 */


int fvgettr(FILE *fp, segy *tp)
   ...
```

Of course, now that `fvgettr` has been used in two codes, it should be extracted as a library function and we should make a convenience macro `vgettr` for the case of standard input. But these are secondary considerations that don't arise for most applications.

For any new SU code, one should provide an example shell program to show how the new code is to be used. Here is such a program for X Windows graphics:

```
#! /bin/sh
# Trivial test of suvlength with X Windows graphics
# Use same graphics set-up as in demos (e.g. demos/Sorting_Traces/Demo/Xsort)

GRAPHER=xgraph
IMAGER=suximage
WIGGER=suxwigb
WIDTH=700
HEIGHT=900
WIDTHOFF=50
HEIGHTOFF=20

>tempdata
>vdata
suplane >tempdata  # default is 32 traces with 64 samples per trace
suplane nt=72 >>tempdata
suvlength <tempdata ns=84 |
sushw key=tracl a=1 b=1 >vdata

# Plot the data
$WIGGER <vdata \
```

```
        perc=99 title="suvlength test"\
        label1="Time (sec)" label2="Traces" \
        -geometry ${WIDTH}x${HEIGHT}+${WIDTHOFF}+${HEIGHTOFF} \
        wbox=$WIDTH hbox=$HEIGHT xbox=$WIDTHOFF ybox=$HEIGHTOFF &


# Remove #comment sign on next line to test the header
#sugethw <vdata tracl ns | more

# Clean up
rm tempdata vdata
```

and here is the PostScript equivalent:

```
#! /bin/sh
# Trivial test of suvlength with PostScript graphics
# Use same graphics set-up as in demos (e.g. demos/Sorting_Traces/Demo/PSsort)
# set PostScript Previewer here if environment variable PSPREVIEWER not set
VIEWER=$PSPREVIEWER

GRAPHER=psgraph
IMAGER=supsimage
WIGGER=supswigp

>tempdata
>vdata
suplane >tempdata  # default is 32 traces with 64 samples per trace
suplane nt=72 >>tempdata
suvlength <tempdata ns=84 |
sushw key=tracl a=1 b=1 >vdata

# Plot the data
$WIGGER <vdata perc=99 title="suvlength test"\
        label1="Time (sec)" label2="Traces" | $VIEWER

# Remove #comment sign on next line to test the header
#sugethw <vdata tracl ns | more

# Clean up
rm tempdata vdata
exit
```